2004－QAI－13　(6)
2004／10／28

VOD配信に適したPeer-to-Peerファイル共有システム

戸田　智雄†　　生野　徳彦†　　寺岡　文男†

† 慶應義塾大学大学院理工学研究科 寺岡研究室
〒 223-8522 神奈川県横浜市港北区日吉 3-14-1
E-mail: †{com,narupy}@tera.ics.keio.ac.jp, ††tera@ics.keio.ac.jp

あらまし　本稿では、DHT(Distributed Hash Table) を用いて構築した Peer-to-Peer ネットワーク上で容量の大きなファイルの配信を行うことで、配信元の帯域が狭い場合でも大規模な配信を行う事が可能なシステム TomCast を提案する。TomCast ではファイルの受信中にも他のノードにファイルを送信できる方式をとることで、容量の大きなファイルの配信効率を高め、TomCast ではファイルを必要としていないノードにかかる配信の負荷を最小限に抑えた。さらに、システム上にファイルが存在する場合は高い確率ですべてのノードがファイルを参照できるシステムとした。
キーワード　Peer-to-Peer DHT VOD

TomCast: A Peer-to-Peer System Suitable for Delivering Large Files

Tomotake TODA†, Naruhiko IKUNO†, and Fumio TERAOKA†

† Teraoka Laboratory, Graduate School of Science and Technology, Keio University
3-14-1 Hiyoshi Kohoku-ku, Yokohama-shi, Kanagawa-ken 223-8522, Japan
E-mail: †{com,narupy}@tera.ics.keio.ac.jp, ††tera@ics.keio.ac.jp

Abstract　This paper proposes a scalable peer-to-peer system called TomCast which is constructed with DHT(Distributed Hash Table). On this system, files are not stored on nodes that do not need the files. TomCast avoids concentrating of the load for processing DHT queries and delivering files at a specific nodes. A node can deliver files to other nodes even when the node is receiving the files, so TomCast can spread files efficiently.
Key words　Peer-to-Peer DHT VOD

1. はじめに

現在では、インターネットの広帯域回線が各家庭まで引かれるようになっている。さらに、キャリア間のシェア獲得競争の激化によって、ADSL の高速化や、FTTH の普及が急速に押し進められている。

その一方で、広帯域の回線を活用するコンテンツはほとんど現れていない。広帯域の回線を活用できるコンテンツの一つとして、解像度の高い映像のインターネット配信が挙げられるが、現状では各ユーザーの持つ帯域を十分に活用できるほどの、高画質な映像の配信はほとんど行われていない。特に、需要が大きいコンテンツほど視聴に耐えない低画質で提供されている。

これは、ユーザーの回線が急速に高速化している一方で、コンテンツ提供者の持つ回線の増強が追い付いていないことが原因として挙げられる。現在でも、需要のあまりないコンテンツは、ある程度高画質で提供することが可能である。しかし需要の大きなコンテンツは、十分な資源を持ったコンテンツ提供者が提供する時でさえ、提供者の回線の不足のために視聴者を限定しなければ、高画質なコンテンツを提供できていない。

以上の問題は、現在のサービス提供形態が、負荷を 1 点または特定の点に集中させてしまう、クライアントサーバー型の方式で実現されていることによって生じている。負荷が特定の場所に集中するサービスの提供形態では、ユーザー数の増加に応じてコンテンツ提供のための資源を増やし続けなければならない。よって、コンテンツ提供者側のリソースの不足という問題を解決するためには負荷が特定の点に集中しないサービスの提供形態を用いる必要がある。1 点に集中する負荷を軽減する既存の手法としてはマルチキャスト技術があるが、VOD(Video on Demand) のようなユーザーからの要求が非同期に発生するサービスでは、一斉にデータを送信する手法であるマルチキャスト技術を用いて、サーバーにかかる負荷を軽減することは困難である。

そこで、本研究ではシステム全体に負荷を分散し、需要の大きい大容量のコンテンツを非同期に提供することが可能なシステムを提案する。負荷分散を行う手法として、現在急速に利用が拡大している Peer-to-Peer の技術を用いる。Peer-to-Peer

ネットワークを構築し、ユーザー同士でコンテンツの授受を行うことで各ユーザーの持つ余剰帯域を活用し、システム全体への負荷分散を図る。

また、VOD では受信に非常に長い時間がかかる、容量の大きなコンテンツが多いため、ファイルを受信し終ってから他のユーザーに送信するシステムでは、得たデータを他のユーザーに渡すまでに長い時間を要してしまう。そこで、ファイルの受信中でも他のユーザーにファイルを送信可能なシステム、TomCast を提案する。

2. TomCast の概要

TomCast では、TomCast システムに参加しているノード間でファイルを授受することで、送信元が直接ファイルを送信するノード数を減らし、負荷の分散を行っている。ファイルを受信したノードは自身がファイルを保持しているという情報を Peer-to-Peer 型のネットワーク上に保存する。ファイルを保持するノードの情報を保存するために、DHT(Distributed Hash Table) [1] [2] [3] [4] という技術を用いて Peer-to-Peer ネットワークを構築する。DHT とは、オーバーレイネットワークという IP ネットワーク上に作られる仮想的なネットワークを構築する技術の一つである。TomCast では Kademlia [2] の仕組みを元に設計を行っているが、他の DHT を用いても本方式と同様の機能を持ったシステムを構築することは可能である。

また、TomCast ではファイル全体の受信を完了していないノードが他のノードに対してファイルを送信する事ができるようにするために、ファイルをいくつかのブロックに分割して扱う。

TomCast におけるファイルの授受の概要を図 1 に示した。簡単のために、図 1 で DHT は仮想的なストレージのように表記してあるが、実際は TomCast に参加するノードすべてで構築された Peer-to-Peer 型のネットワークである。図 1 を用いて TomCast の概要を説明する。この図で配信対象となるファイルは sample.file である。

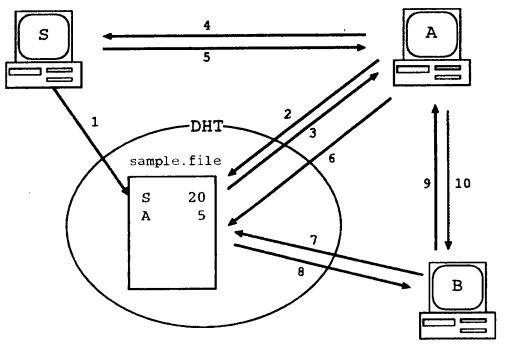

図 1 TomCast の概要

（1） ファイルの最初の配信元であるノード S が、DHT 上に sample.file を保持するノードのリストを作成する。そして、このリストに自身へのポインタと保持するブロック数 (ここでは、ファイル全体のブロック数となる) を保存する。

（2） sample.file を必要とするノード A が DHT に対してリストを要求する。

（3） ノード A が DHT からリストを得る。

（4） ノード A が、取得したリストを元に、S に対し直接ファイルを要求する。

（5） S から A に対し直接ファイルが送信される。

（6） 受信中の A が自身へのポインタと受信ブロック数を、DHT 上の sample.file を保持するノードのリストに保存する。

（7） さらに、sample.file を必要とするノード B が DHT に対してリストを要求する。

（8） ノード B が DHT からリストを得る。

（9） ノード B が、取得したリストを元に、ノード A に対して直接ファイルを要求する。

（10） ノード A はファイルの受信を継続し、さらにノード B に対してファイルを送信する。

3. 関連研究

3.1 DHT の概要

DHT(Distributed Hash Table) とは、IP ネットワーク上に作られる仮想的なネットワークであるオーバーレイネットワークの構築技術の一つである。このオーバーレイネットワーク上では、データの保存やルーティングが行われる。

DHT では DHT 上に保存されるデータ、DHT に参加するノードそれぞれについて、同じ ID 空間上の ID を付ける。この ID はデータに関連づけられたキーワードや、ノードへのポインタを Hash 関数にかけることによって生成する。本論文中ではいくつかの DHT で使われているように、DHT 上に保存されたデータを *value*、そのデータの ID を *key* と呼び、ノードに付けられた ID を *NodeID* と呼ぶ。

DHT では *value* を保存するノードは *key* と *NodeID* によって定まる。これを定める際には、*key* と *NodeID* との差や排他的論理和の大小等を用いる。*value* を保存するノードは、新しいノードの DHT への参加や、ノードの離脱によって替わる。本提案では、Kademlia を主眼に置いて設計を行った。Kademlia では、ルーティングや、*value* を保持するノードの決定に排他的論理和を用いる。*key* との排他的論理和が最も小さい *NodeID* を持つ k 個のノード (k はシステム全体で定められた値) が、その *key* に対応する *value* を保持する。また、ルーティングにかかるホップ数は $O(\log N)$ となる。

3.2 DHT を使ったファイルの配信

DHT を用いてファイルを配信する際には、*value* として配信対象のファイルを保存する手法や、*value* に配信対象のファイルを持つノードへのポインタ (IP アドレスとポート番号等) を保存するといった手法が考えられる。

value として配信対象となるファイルを保存した場合は、いくつかの DHT の仕組みがそれぞれに持っているキャッシュの機構によって、需要の大きなファイルを保存するノードの数が増加し、負荷が分散する。しかし、*value* として容量の大きなファイルを保存すると、DHT へのノードの参加や離脱によって *value* を保存するノードが替わった際に、非常に大きなファ

イルの移動コストがかかってしまう。また、*value* を保持するノードは Hash 関数によってランダムに定められるため、あるノードの多くの帯域とストレージを、そのノードが必要としていないファイルのために消費することになってしまう。これは、ユーザーの DHT への参加意欲を減少させることにもなる。

これに対し、*value* としてファイルを持つノードへのポインタを保存した場合は、ノードの参加や離脱に伴うデータの移動は少なくて済む。また、あるファイルを必要としないノードが、そのファイルのために大きな帯域とストレージを消費する事もない。しかし、この方式ではポインタの先にあるノードに負荷が集中してしまう。この負荷を分散するためには、ファイルを受信したノードへのポインタを次々と *value* に書き加える必要がある。しかし、ファイルを受信したノードのポインタのリストを value として保存しても、そのエントリの頻繁な追加や削除は通常の DHT では難しい。これは、*value* を取得し、ポインタのリストに対し自身へのポインタを追加し、そのリストを *value* として保存する、という手順をそれぞれのノードが行っていては同期を取ることが難しいためである。また、このポインタのリストへのアクセスが集中すれば、リストを持つノードに負荷が集中してしまうという問題も生じる。

3.3 Coral

前節までに挙げられた問題に対処するために、Coral [5] という手法がある。これはファイルを持つノードへのポインタを DHT の value として保持する方法に対して、改良を加えたものである。Coral では *value* として保存されたポインタのリストに対して、エントリを追加する機能を付け加えている。これによって、*value* の同期を考えることなくエントリの追加を行うことができる。さらにリストのエントリ数に上限を設けている。上限を超えるエントリを保存する場合には、エントリを追加しようとしているノードから、リストを保存するノードまでの DHT 上での経路の上で、最後の 1 ホップを手前に戻ったところにあるノードに *value* を保存する。1 ホップ戻った先のノードのエントリ数も上限値に達している場合は、さらに 1 ホップ戻って保存する。このようにすることで、リストを保存しているノードへの負荷を減らしている。Coral ではエントリの保存先のノードがその時々の経路によって定まるため、エントリを追加したノード自身がその更新や削除を行うことが困難である。そのため、ファイルの受信中にその受信状況を頻繁に更新するといった利用はできない。また、Coral は削除の機構を持たないため、一定時間経過したエントリは無条件に削除してリスト中の有効なエントリの割合を高めている。これは、有効なノードを減少させてしまう。

これに対し、TomCast では、エントリの存在する場所を常に特定できるようにすることで、ファイルの受信状況を常に更新できるようにした。また、ファイルを保持しているノードへのポインタをシステム上から失わないようにした。

4. TomCast の仕組み

4.1 TomCast のノードリスト

TomCast ではファイルを保持するノードへのポインタと、保持するブロックの数のリストを *value* として保存する。このリストをノードリストと呼ぶ。そして、ファイルを受信したノードが次々にノードリストにエントリを追加、更新ができるようにするために、DHT 上に保存されたノードリストに対してエントリを追加する機能を加えた。さらに、このノードリストのエントリ数には上限を設け、エントリ数が上限を越えた場合はノードリストの数を増やしていく。しかし、DHT では *key* と *value* は 1 対 1 に対応するため、ノードリストの数を増やすためには、*key* を生成する元となるキーワードを変更する必要がある。そこで、TomCast では次のようにして新たなリストを生成する。以下、hash(X) で X を Hash 関数にかけた結果を示す事とする。まず、最初のノードリストの *key* を生成するためのキーワードはファイル名を用いる。key = hash(ファイル名) である。この最初のノードリストのエントリ数が上限に達した時は、hash(追加しようとしているエントリのポインタ) の最上位ビットをファイル名に付け加え、キーワードを生成する。そして、このキーワードを Hash 関数にかけて *key* を生成し、その *key* に対応する *value* として新しいノードリストを保存する。さらに、このリストもエントリ数の上限に達した場合は、hash(追加しようとしているエントリのポインタ) の上位 2 ビットをファイル名に加えてキーワードを生成する。図 2 に新たな key の生成例を示した。図 2(a) で新たに加えられるエントリは、IP アドレス xxx.xxx.xxx.xxx、ポート番号 1234 というポインタを持つノードである。このノードは sample.file というファイルを受信し終わったか受信中である。そのため、hash(sample.file) を key として保存されたノードリストに、自身へのポインタを加えようと試みる。しかし、エントリ数が上限に達していたため、自身へのポインタを Hash 関数にかけて、Hash 値 10010... を得た。そして、その上位 1 ビットをファイル名に加えた文字列 sample.file_1 を得る。そして、hash(sample.file_1) を *key*(*key* の値は 11001...) として、自身のエントリを新たなノードリスト

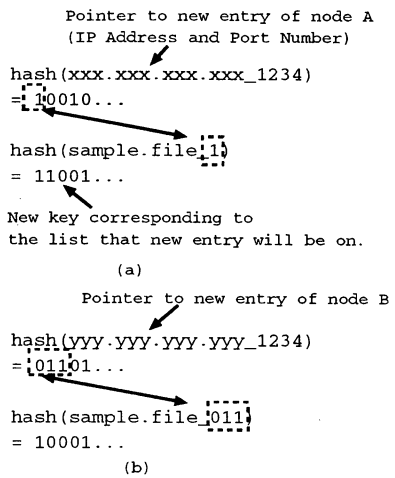

図 2 ノードリストの生成

に保存する。hash(sample.file_1) を *key* として保存されたノードリストを、ノードリスト sample.file_1 というように呼ぶ。図 2(b) は IP アドレス yyy.yyy.yyy.yyy、ポート番号 1234 をポインタとして持つノードを新たにエントリとして加えようとしている。ポインタを Hash 関数にかけた結果は 01101... である。また、すでにファイル名に 0,01 を加えた文字列を元に作成されたノードリスト sample.file_0, sample.file_01 が存在し、ノードリスト sample.file_01 のエントリ数が上限値に達している。この場合は、sample.file という文字列に、hash(ポインタ) の上位 3 ビット、011 をファイル名に加えた文字列 sample.file_011 をキーワードとして生成されたノードリスト、つまりノードリスト sample.file_011 にエントリを加える。このように作成されたノードリストは、図 3 に示したような木構造を取る。図 3 でノードリスト sample.file_10 には、hash(IP アドレス_ポート番号)=10... となるエントリが、ノードリスト sample.file_1 には hash(IP アドレス_ポート番号)=1... となるエントリが、というように保存されている。

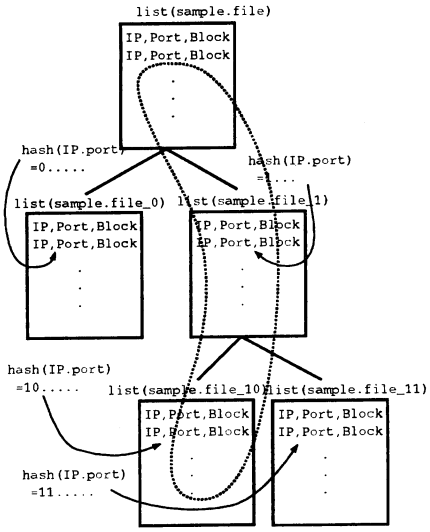

図 3　ノードリスト群の構造

ここで、ノードリストを作成する際に用いたキーワードに付け加えた hash 値のビット数が少ないノードリストを上位のノードリスト、ビット数の多いノードリストを下位のノードリストと呼ぶ。

このような方法でリストを増加させる事によって、あるエントリが存在する可能性のあるノードリストの数を制限することが可能である。キーワード生成の際に加えるビット数の最大値＋1 個のノードリストのどれかには、必ずある特定のノードのエントリが存在することがわかる (システム上にそのエントリが存在する場合)。つまり、木の根から葉に向かう一筋の経路上のどこかにエントリが存在することがわかる。図 3 では、点線で囲ったノードリストが、hash(IP アドレス_ポート番号)=10... となるエントリが存在する可能性のあるノードリストを示している。このことによって、後述するノードリストの統合を行い、エントリの保存されるノードリストが変わった際にも、自身のエントリの位置を特定することができるため、そのエントリの更新が可能となる。

4.2　ノードリストの参照

あるファイル sample.file を必要としているノードが、そのファイルを保持しているノードが載っているノードリストを取得する時は、まず自身へのポインタを Hash 関数にかける。次に、得られた Hash 値の上位何桁かをファイル名に付け加えた文字列を Hash 関数にかけることによって *key* を得る。この *key* について DHT 上で参照を行い、ノードリストを得る。ノードリストが得られない場合は付け加える桁数を減らすことで最終的にノードリストを得る事ができる。このようにしてノードリストを取得することで、取得したノードリストが自身のポインタを加えるべきノードリストとなるため、DHT 上で *value* を持つノードを特定する回数を減らすことができる。

また、ノードリストを取得する際に付け加えるビット数は次のように定める。ノードリスト作成の際に使用するキーワードの作成時に、ファイル名に付け加えるビット列のビット数の最大値に、1 を加えた数を H とする。そして、n 回目の参照の際に付け加えるビット数 $h_n - 1$ を

$$h_{n+1} = h_n \pm \tfrac{1}{2}(h_n - h_{n-1}),\ \text{if } n \geqq 2$$
$$h_1 = \tfrac{H}{2}$$
$$h_0 = 0$$

とする。付け加えるビット数が h_n ではなく、$h_n - 1$ なのは最初のノードリストを生成する際には、キーワードにファイル名だけを用いて、ビット列は加えないためである。± は、参照してノードリストを得た場合は +、得られなかった場合は − を取る。この参照の様子を図 4 に示した。図は hash(自身へのポインタ)=1000111... となるノードの参照手順で、H の値は 8 である。図中の四角はノードリストを表している。ここで、必要としているファイルは sample.file であるとする。図 4 で $\frac{H}{2}$ は 4 であるので、hash(自身へのポインタ) の上位 3 ビットをファイル名に付けて *key* を生成する。*key*=hash(sample.file_100) となる。この *key* について DHT 上で参照を行い、ノードリスト sample.file_100 を得た。次に、5 ビットを加えノードリスト sample.file_10001 を参照したが、存在しなかったため、ノードリスト sample.file_1000 を参照し、ノードリスト sample.file_1000 がこのノードが参照する事のできるノードリストのうち、最も下位のノードリストであることがわかった。

このように参照を行うことで、$\log H$ 回の参照で、参照対象となるノードリストのうち、最も下位のものを得ることができる。最も下位のノードリストを取得する理由は、下位のノードリストほど参照をする可能性のあるノードが少ないため (ノードリスト sample.file_0110 を参照する可能性のあるノードは、自身のポインタを Hash 関数にかけた結果が 0110... になるノードだけとなるが、sample.file_0 を参照する可能性のあるノードは自身のポインタを Hash 関数にかけた結果が 0... となるノードになる。)、そのノードリストを保持するノードへの負荷が集中しづらいからである。仮に、すべてのノードが最も上位の

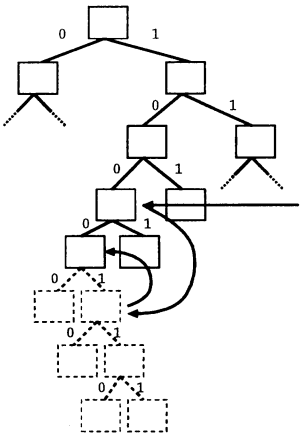

図 4　ノードリストの参照手順

ノードリストを取得すると、最上位のノードリストを保持するノードに負荷が集中してしまう。

4.3　ノードリストの統合

TomCast では、TomCast からの離脱によって無効になったノードのエントリを削除することができる。そして、削除によってエントリ数が少なくなったノードリストを、上位のノードリストに統合する事ができる。このことによって、有効なエントリを上位のノードリストに集めることができる。下位のノードリストは利用できるノードが限られているため、下位のノードリストの一部にだけ有効なエントリが存在する状況では、システム上に利用できるファイルが存在するにもかかわらず、一部のノードしかそのファイルを利用できないという状況が発生してしまう。上位のノードリストに有効なエントリを集めることは、こういった状況を防ぐために有効である。

ノードリストの統合は、あるリストのエントリ数が閾値より少なくなり、なおかつ下位のリストが存在する場合に行われる。エントリ数が閾値より少なくなったノードリストの 1 つ下位にある 2 つのノードリストから、エントリを上位のノードに移すことで統合を行う。

5.　TomCast メッセージの詳細

TomCast で用いる主なメッセージは、*add_entry, delete_entry, get_list, ping_f* の 4 メッセージと Kademlia のメッセージである、*find_node, ping* である。すべてのメッセージには *RPC_ID* というランダムな数値が毎回付けられる。あるメッセージの応答としてメッセージを送信する場合には、送られてきた *RPC_ID* をそのまま付けて返す。このことによって、メッセージの詐称を最低限防止している。*find_node* メッセージは、ある *key* に対応する *value* を保存すべき k 個のノードを特定するためのメッセージである。*ping* メッセージは送信先のノードが存在するかを確認するメッセージである。

5.1　get_list

get_list RPC_ID list_name

get_list メッセージは、*find_node* メッセージによって、あるノードリストを保存すべきノードを特定した後に、その特定されたノードに対して送信する。受信したノードは引数で指定されたノードリストを保持していればノードリストを返す。

5.2　ping_f

ping_f RPC_ID file_name

ping_f メッセージは、送信先のノードがあるファイルを保持しているかを確認するためのメッセージであり、引数として *RPC_ID* とファイル名を取る。*ping_f* メッセージを受信したノードは、引数で指定されたファイルを保持していれば、このメッセージに応答する。

5.3　add_entry

add_entry RPC_ID node_list_name IP_Address Port_Number Blocks ...

add_entry メッセージはノードリストにエントリを追加するためのメッセージであり、複数のエントリの追加要求を 1 つのメッセージで要求できる。引数として *RPC_ID* の他に、追加するノードリストの名前、追加するノードのポインタと保持するブロック数を列挙したものを取る。*add_entry* メッセージを受信したノードは、追加するノードリストに空きがある場合は、追加するポインタに向かって、*ping_f* メッセージを送信し対象となるファイルを保持しているかを確認する。対象となるファイルを保持していることを確認した場合は、エントリを追加する。そして、追加完了後 *add_entry* の送信元に追加したエントリを伝える。

5.4　delete_entry

delete_entry RPC_ID node_list_name IP_Address Port_Number ...

delete_entry メッセージはノードリストからエントリを削除するためのメッセージであり、複数のエントリの削除要求を 1 つのメッセージで要求できる。引数として *RPC_ID* の他に、エントリを削除するノードリストの名前、削除するノードのポインタを列挙したものを取る。*delete_entry* メッセージを受信したノードは引数で指定されたノードに対して *ping_f* メッセージを送り、一定時間応答がなかった場合はエントリを削除する。一定時間後に、削除をしたエントリを *delete_entry* の送信元に返す。

6.　評　　価

6.1　セキュリティ

TomCast では、ノードリストへのエントリの追加の際に、ノードリストを保持するノードが実際にファイルを保持しているかを、*ping_f* メッセージを用いて確認する。これによって、悪意あるノードが攻撃対象とするノードのポインタを大量に加える事による DoS 攻撃を防ぐことができる。また、ランダムな *RPC_ID* をメッセージの引数として加えることによって、IP アドレスを詐称して *ping_f* メッセージに対する応答を行うことを難しくしている。ノードリストを保持するノードは Hash 関数によって定まり、また 1 つのノードリストを保持するノードは複数あるため、悪意あるユーザーが意図的にノードリスト

を保持するノードすべてを支配することは難しい。

また、エントリ削除の際にもノードリストを保持するノードが、本当に無効なエントリであるかを確認するため、悪意あるユーザーがすべてのエントリを削除してシステムを破壊する事を防ぐことができる。

一方、TomCast では入手したファイルの安全性や正当性を確認することはできない。ノードリストに記載されたノードが他のファイルを偽ってウイルスを送信したとしてもそれを判別する機構は備わっていない。この点は今後の課題である。

6.2 スケーラビリティ

TomCast の各操作に必要なメッセージ数を表 1 にまとめた。表中の k は Kademlia 上で、同じ *value* を保持するノードの数である。また、*Lookup* は Kademlia 上で *value* を持つべき k 個のノードを特定するという操作を行う回数である。1 回の *Lookup* に必要なメッセージ数は、N を TomCast に参加している全ノードとして $O(\log N)$ となる。なお、すべてのメッセージにはその応答となるメッセージが存在するため、応答のメッセージ数を合わせると実際のメッセージ数は表中の値の 2 倍となる。

表 1 TomCast の各操作に必要なメッセージ数

操作の種類	*Lookup*	メッセージ数
ノードリストの取得	$\log H$	$\log H$ * *get_llist*
エントリの追加	0 以上 $\log H$ 未満 (通常は 0)	*get_list* *add_entry* * k *ping_f* * k * 追加エントリ数
エントリの更新	0 以上 $\log H$ 未満 (通常は 0)	*get_list* *add_entry* * k *ping_f* * k
エントリの削除	0	*delete_entry* * k *ping_f* * k * 削除エントリ数
ノードリストの統合	1	*add_entry* * k * 2 *delete_entry* * k *ping_f* * k * 削除エントリ数 *ping_f* * k * 追加エントリ数

- ノードリストの取得

ノードリストを取得する際には、前述のように $\log H$ 回のリストの参照が必要となる。

- エントリの追加

エントリの追加を行う際は、追加を行うノードがファイルの取得の際に利用したリストが、自身のポインタを保存すべきノードリストであるので、ノードリストを保持するノードを毎回特定し直す必要はない。そのため、*Lookup* は 0 回のことが多い。ただし、ノードリストが統合されていた場合は、より上位のリストを得る必要があるため、*Lookup* が必要となる。また、*add_entry* メッセージは value を保存する k 個のノードすべてに送信する必要があるため、*add_entry* * k となっている。

- エントリの更新

エントリの更新は、エントリの追加とほぼ同様であるが、同時に複数のエントリを更新することがない点だけが異なっている。

- エントリの削除

エントリの削除の際に必要なメッセージは、削除要求としての *delete_entry* メッセージと、その正否を返すメッセージ、削除の確認を行う *ping_f* メッセージとその返答である。削除を要求するノードは、すでに削除対象のノードが含まれるノードリストを保持するノードを特定しているため、*Lookup* は必要ない。

- ノードリストの統合

ノードリストの統合を行うノードは、エントリの不足しているノードリストと、さらにそのノードリストの 1 つ下位のノードリストを 1 つを特定している。そのため、2 つある下位のリストのうち、もう 1 つのリストを特定すればよい。よって、*Lookup* は 1 となる。3 つのノードリストを取得した後、下位のノードリストを保持するノードには *delete_entry* メッセージを送り、上位のノードリストを保持するノードには *add_entry* メッセージを送る。さらに、そのそれぞれについて、確認のための *ping_f* メッセージが必要となる。

表 1 から、TomCast の操作において、メッセージ数が TomCast 上のノード数 N に依存するのは、Kademlia 上での *Lookup* だけである。よって、TomCast のスケーラビリティは *Lookup* に必要となるメッセージ数に依存し、そのメッセージ数は $O(\log N)$ となることがわかる。

7. ま と め

TomCast は Peer-to-Peer ネットワークを構築することによって、負荷を分散させ、スケーラビリティを維持した上で、容量の大きなファイルを、ファイルの配布元のリソースが少なくても配布することのできるシステムを提案した。また、ファイルの受信中に受信中のファイルを他のノードに送信可能なシステムを構築することによって、ファイルの配布効率を高めた。さらに、TomCast ではファイルを必要とするノードが、そのファイルの配布によって生じる負荷の多くを負担するため、ネットワークに参加するためだけに必要なコストが少なくて済むことで、ユーザーのネットワークへの参加意欲を妨げないようなシステムを提案した。

文 献

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan: "Chord: A scalable peer-to-peer lookup service for internet applications", Proceedings of SIGCOMM 2001, pp. 149–160 (2001).

[2] P. Maymounkov and D. Maziéres: "Kademlia: A peer-to-peer information system based on the xor metric", Proceedings of IPTPS02 (2002).

[3] A. Rowstron and P. Druschel: "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems", Proceedings of Middleware 2001 (2001).

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummdadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao: "Oceanstore: An architecture for global-scale persistent storage", Proceedings of ASPLOS 2000 (2000).

[5] M. J. Freedman and D. Maziéres: "Sloppy hashing and self-organizing clusters", Proceedings of IPTPS 03 (2003).